



Dotnet France  
Technologies Sharepoint, SQL Server & .NET

Association Dotnet France

# ADO .NET : utilisation des transactions

*Version 1.0*

Harold CUNICO



James RAVAILLE

<http://blogs.dotnet-france.com/jamesr>

# Sommaire

---

1	Introduction.....	3
1.1	Présentation .....	3
1.2	Principes d'exécution d'une transaction.....	3
1.3	Présentation de la base de données .....	4
1.3.1	Création de la base de données <i>DotnetFranceA</i> .....	4
1.3.2	Création de la base de données <i>DotnetFranceB</i> .....	4
2	Les transactions locales.....	6
2.1	Création d'une transaction locale .....	6
2.1.1	Présentation .....	6
2.1.2	Mise en œuvre.....	6
2.2	Les niveaux d'isolations.....	8
2.2.1	Présentation .....	8
2.3	Mise en œuvre.....	9
2.3.1	Présentation du formulaire .....	9
2.3.2	Gestion des niveaux d'isolation de données.....	9
2.3.3	Gestion des données .....	11
2.3.4	Exécution de l'application .....	15
3	Les transactions distribuées .....	17
3.1	Présentation .....	17
3.2	Mise en œuvre.....	18
4	Conclusion.....	21

## 1 Introduction

### 1.1 Présentation

Une transaction est un ensemble d'opérations réalisées sur une base de données, exécutée de manière unitaire. En tant qu'unité, ces opérations sont validées uniquement si toutes ont été exécutées avec succès (en application du principe « tout ou rien »). Il existe quatre propriétés connues sous le nom ACID qui définissent une transaction :

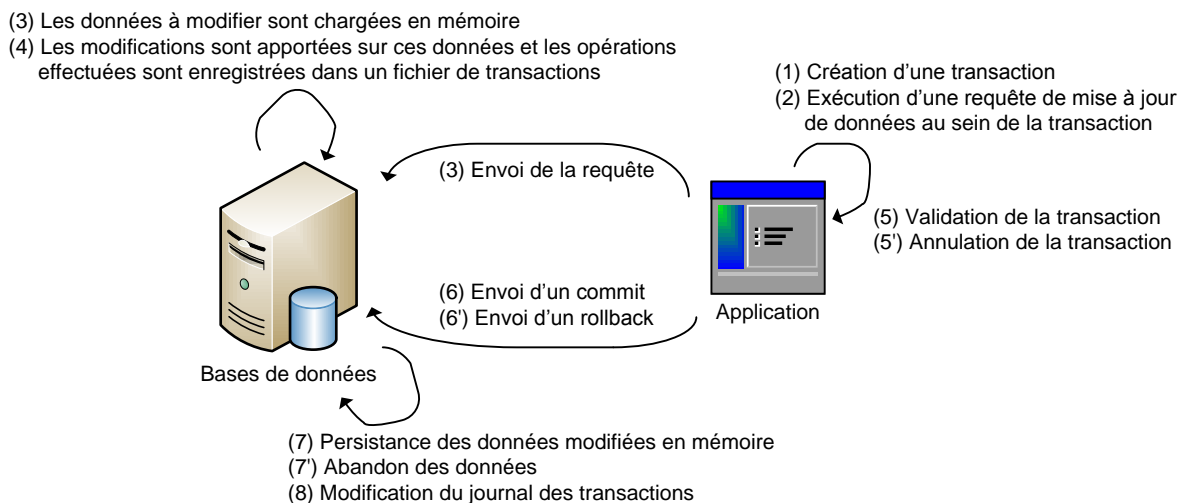
- **Atomicité** représente l'intégralité des opérations effectuées par une transaction : soit elles sont validées dans leur ensemble (*commit*), soit la transaction est annulée (*rollback*).
- **Cohérence** représente l'intégrité de la base de donnée, que la transaction est étai validé ou annulé, la base de donnée doit rester intègre.
- **Isolation**, une transaction est indépendante d'une autre transaction. Les transactions ne peuvent interférer entre elles.
- **Durabilité**, les données sont préservées une fois la transaction achevée.

Ces propriétés démontrent tout l'intérêt des transactions quand plusieurs lignes (row) d'une table doivent être modifiées. Quand une modification sur une table entraîne des modifications sur d'autres tables (contrainte de clé primaire). Ou bien encore dans tout autre cas qui porterait atteinte aux propriétés ACID.

Il existe deux types de transactions en ADO.net, les transactions locales et les transactions distribuées. Les transactions locales sont des transactions simples qui effectuent des opérations sur une seule ressource (par exemple une base de données). Les transactions distribuées quand à elles, sont des transactions qui utilisent de nombreuses ressources.

### 1.2 Principes d'exécution d'une transaction

Le schéma ci-dessous vous expose le principe d'exécution d'une transaction :



Les choix entre *i* et *i'* sont réalisés en fonction des choix de l'utilisateur, dès l'étape 5.

### 1.3 Présentation de la base de données

Dans les cas pratiques présentés dans ce cours, nous allons dans un premier temps utiliser une base de données SQL Server 2008 nommée *DotnetFranceA*. Puis, lorsque nous aborderons les transactions distribuées, nous utiliserons une base de données supplémentaire nommée *DotnetFranceB*.

#### 1.3.1 Création de la base de données *DotnetFranceA*

La base de données *DotnetFranceA* ne contient qu'une seule table, nommée *Stagiaire*. Voici un script SQL, permettant de créer cette table :

```
// SQL

CREATE TABLE [dbo].[Stagiaire] (
    [id] [int] NOT NULL,
    [nom] [varchar] (50) NULL,
    [prenom] [varchar] (50) NULL,
    [adresse] [varchar] (50) NULL,
    [telephone] [varchar] (50) NULL,
    [mail] [varchar] (50) NULL,
    [information] [varchar] (500) NULL,
    CONSTRAINT [PK_Stagiaire] PRIMARY KEY CLUSTERED
    (
        [id] ASC
    ) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY],
    UNIQUE NONCLUSTERED
    (
        [id] ASC
    ) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
GO
```

Et voici un autre script permettant d'alimenter cette table :

```
// SQL

INSERT [dbo].[Stagiaire] ([id], [nom], [prenom], [adresse], [telephone], [mail], [information]) VALUES (1, N'PALUDETTO', N'Damien', N'0', N'0', N'0', N'0')
INSERT [dbo].[Stagiaire] ([id], [nom], [prenom], [adresse], [telephone], [mail], [information]) VALUES (4, N'SAGARA', N'Sousuke', N'0', N'0', N'0', N'0')
GO
```

#### 1.3.2 Création de la base de données *DotnetFranceB*

La base de données *DotnetFranceB* ne contient qu'une seule table, nommée *Manager*. Voici un script SQL, permettant de créer cette table :

```
// SQL

CREATE TABLE [dbo].[Manager] (
    [id] [int] NOT NULL,
    [respId] [int] NULL,
    [nom] [varchar] (50) NULL,
    [prenom] [varchar] (50) NULL,
    [adresse] [varchar] (50) NULL,
    [telephone] [varchar] (50) NULL,
    [mail] [varchar] (50) NULL,
    [information] [varchar] (500) NULL,
    CONSTRAINT [PK_Manager] PRIMARY KEY CLUSTERED
(
    [id] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY],
UNIQUE NONCLUSTERED
(
    [id] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY =
OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
GO
```

Et voici un autre script permettant d'alimenter cette table :

```
// SQL

INSERT [dbo].[Manager] ([id], [respId], [nom], [prenom], [adresse],
[telephone], [mail], [information]) VALUES (1, 1, N'Ravaille', N'0',
N'0', N'0', N'0', N'0')
INSERT [dbo].[Manager] ([id], [respId], [nom], [prenom], [adresse],
[telephone], [mail], [information]) VALUES (2, 2, N'Vasselon', N'0',
N'0', N'0', N'0', N'0')
INSERT [dbo].[Manager] ([id], [respId], [nom], [prenom], [adresse],
[telephone], [mail], [information]) VALUES (3, 3, N'Hollebecq', N'0',
N'0', N'0', N'0', N'0')
INSERT [dbo].[Manager] ([id], [respId], [nom], [prenom], [adresse],
[telephone], [mail], [information]) VALUES (4, 4, N'Dominiquez', N'0',
N'0', N'0', N'0', N'0')
GO
```

## 2 Les transactions locales

### 2.1 Création d'une transaction locale

#### 2.1.1 Présentation

Pour créer une transaction en ADO.net, on utilise une classe dérivant de la classe *DbTransaction*. Cette classe appartient à l'espace de nom *System.Data.Common* du composant *System.data.dll* du Framework .NET. Cette classe expose les méthodes *BeginTransaction*, *Commit* et *Rollback*, essentielles dans la mise en œuvre des transactions. En fonction des classes d'accès aux données utilisées, vous utiliserez la classe suivante :

- `System.Data.Odbc.OdbcTransaction`
- `System.Data.OleDb.OleDbTransaction`
- `System.Data.OracleClient.OracleTransaction`
- `System.Data.SqlClient.SqlTransaction`

La méthode *BeginTransaction* permet de débiter la transaction. La méthode *Commit* permet de valider les modifications effectuées par les requêtes exécutées au sein de la transaction. La méthode *Rollback* permet d'annuler ces modifications.

#### 2.1.2 Mise en œuvre

Dans l'exemple ci-dessous, nous utilisons une transaction, au sein de la quelle sont exécutées deux requêtes « Insert » sur une base de données SQL Server 2008. Nous utiliserons donc la classe *SqlTransaction*.

Si les deux requêtes s'exécutant au sein de cette transaction sont exécutées avec succès :

- Nous appliquons sur l'objet Transaction la méthode *Commit*, pour valider la transaction.
- Dans une boîte de message, nous afficherons le message « Transaction validée ».

Dans le cas où l'exécution d'une des deux requêtes échoue :

- Nous appliquons sur l'objet Transaction la méthode *Rollback*, pour annuler la transaction.
- Dans une boîte de message, nous affichons le message d'erreur.

Dans notre cas, nous ajoutons un premier Stagiaire. Cet ajout s'exécute normalement. Le second provoque la levée d'une exception, car l'ajout d'un stagiaire avec un identifiant existant provoque une violation d'une contrainte de clé primaire. Ainsi, la transaction est annulée, et aucun ajout n'est effectué en base de données.



```
// C#

string connectionString = "Data Source=NORBERT\\SQLEXPRESS;Initial
Catalog=DotnetFrance;Integrated Security=true";
using (SqlConnection connection = new SqlConnection(connectionString))
{
    connection.Open();
    SqlTransaction transaction = connection.BeginTransaction();
    SqlCommand commande = connection.CreateCommand();
    commande.Transaction = transaction;

    try
    {
        //commande 1
        commande.CommandText = "INSERT Stagiaire (id, nom, prenom, adresse,
telephone, mail, information) VALUES (7, 'DOLLON', 'Julien', '0', '0',
'0', '0')";
        commande.ExecuteNonQuery();

        //commande 2
        commande.CommandText = "INSERT Stagiaire (id, nom, prenom, adresse,
telephone, mail, information) VALUES (4, 'VERGNAULT', 'Bertrand', '0',
'0', '0', '0')";
        commande.ExecuteNonQuery();

        transaction.Commit();
        MessageBox.Show("Transaction validée");
    }
    catch (Exception Ex)
    {
        transaction.Rollback();
        MessageBox.Show(Ex.Message);
    }
    finally
    {
        connection.Close();
    }
}
```



```
// VB .NET

Dim connectionString As String = "Data Source=NORBERT\SQLEXPRESS;Initial
Catalog=DotnetFrance;Integrated Security=true"
Using connection As New SqlConnection(connectionString)
    connection.Open()
    Dim transaction As SqlTransaction = connection.BeginTransaction()
    Dim commande As SqlCommand = connection.CreateCommand()
    commande.Transaction = transaction

    Try
        'commande 1
        commande.CommandText = "INSERT Stagiaire (id, nom, prenom, adresse,
telephone, mail, information) VALUES (7, 'DOLLON', 'Julien', '0', '0',
'0', '0')"
        commande.ExecuteNonQuery()

        commande.CommandText = "INSERT Stagiaire (id, nom, prenom, adresse,
telephone, mail, information) VALUES (4, 'VERGNAULT', 'Bertrand', '0',
'0', '0', '0')"
        commande.ExecuteNonQuery()

        transaction.Commit()
        MessageBox.Show("Transaction validée")
    Catch Ex As Exception
        transaction.Rollback()
        MessageBox.Show(Ex.Message)
    Finally
        connection.Close()
    End Try
End Using
```

## 2.2 Les niveaux d'isolations

### 2.2.1 Présentation

Une application doit souvent gérer de nombreuses transactions simultanément, ainsi que l'accès aux données entre les différentes transactions. Vous trouverez ci-dessous la liste des propriétés de l'énumération *IsolationLevel* associées à l'objet Transaction. Chaque propriété définit un niveau d'accès aux données en cours de modification par d'autres transactions en cours. Les données qui sont en cours de modification sont dites données volatiles.

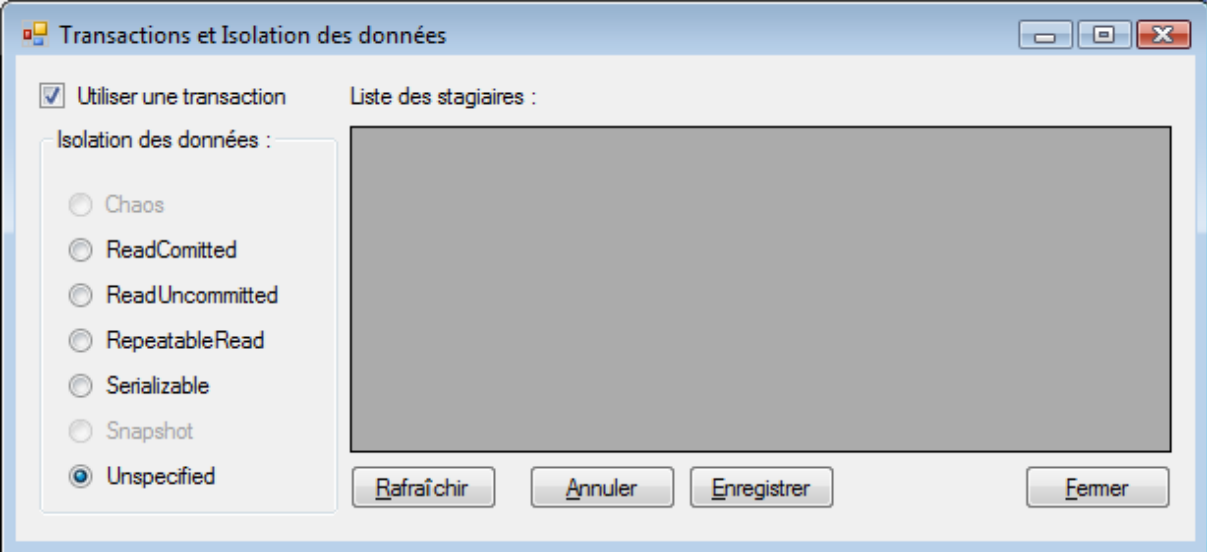
- **Chaos** : les données en attente de transactions très isolées ne peuvent être écrasées.
- **ReadComitted** : les données volatiles ne peuvent pas être lues pendant la transaction, mais peuvent être modifiées.
- **ReadUncommitted** : les données volatiles peuvent être lues et modifiées pendant la transaction.
- **RepeatableRead** : les données volatiles peuvent être lues mais pas modifiées pendant la transaction. De nouvelles données peuvent être ajoutées.
- **Serializable** : niveau d'isolation par défaut. Les données volatiles peuvent être lues mais pas modifiées. De même aucune nouvelle donnée ne peut être ajoutée pendant la transaction.
- **Snapshot** : les données volatiles peuvent être lues. La transaction vérifie que les données initiales n'ont pas changées avant de valider la transaction. Cela permet de régler les problèmes liés à l'accès concurrentiels aux données.
- **Unspecified** : aucun niveau ne peut être déterminé.

## 2.3 Mise en œuvre

Nous allons maintenant créer une petite application, permettant de gérer en mode CRUD (Create, Read, Update, Delete) les données contenues dans la table. Ces données seront lues au sein d'une transaction, puis affichées. Nous aurons la possibilité de choisir un niveau d'isolation des données. Puis, nous lancerons deux instances de notre application, afin de mettre l'évidence l'utilisation des transactions et des niveaux d'isolation.

### 2.3.1 Présentation du formulaire

Le formulaire que nous allons réaliser est le suivant :



Il permet :

- D'afficher la liste des stagiaires, tout en nous laissant la possibilité de verrouiller des données, via l'utilisation d'une transaction. Les niveaux de visibilité Chaos et Snapshot sont grisés, car ils ne sont pas applicables à la base de données SQL Server telle quelle (nommée *DotnetFranceA*).
- D'annuler les modifications de données effectuées dans la grille.
- D'enregistrer les modifications en base de données.

### 2.3.2 Gestion des niveaux d'isolation des données

Nous implémentons l'évènement *CheckedChanged* le contrôle *CheckBox* intitulé « Utiliser une transaction », afin d'activer ou non la liste des niveaux de visibilité :

```
// C#  
  
private void ChkUtiliserTransaction_CheckedChanged(object sender,  
EventArgs e)  
{  
    GbxNiveauxIsolation.Enabled = ChkUtiliserTransaction.Checked;  
}
```

```
// VB .NET

Private Sub ChkUtiliserTransaction_CheckedChanged(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
ChkUtiliserTransaction.CheckedChanged
    GbxNiveauxIsolation.Enabled = ChkUtiliserTransaction.Checked
End Sub
```

Pour gérer les différents niveaux d'isolation nous affectons à chaque bouton radio, un niveau d'isolation de données qui lui est propre, lors du chargement du formulaire. Pour ce faire, nous utilisons la propriété *Tag*, présente sur l'ensemble des contrôles Windows Forms.

```
// C#

private void FrmIsolationDonnees_Load(object sender, EventArgs e)
{
    RbtChaos.Tag = IsolationLevel.Chaos;
    RbtReadComitted.Tag = IsolationLevel.ReadCommitted;
    RbtReadUncommitted.Tag = IsolationLevel.ReadUncommitted;
    RbtRepeatableRead.Tag = IsolationLevel.RepeatableRead;
    RbtSerializable.Tag = IsolationLevel.Serializable;
    RbtSnapshot.Tag = IsolationLevel.Snapshot;
    RbtUnspecified.Tag = IsolationLevel.Unspecified;

    RbtUnspecified.Checked = true;
}
```

```
// VB .NET

Private Sub FrmIsolationDonnees_Load(ByVal sender As System.Object, ByVal
e As System.EventArgs) Handles MyBase.Load
    RbtChaos.Tag = IsolationLevel.Chaos
    RbtReadComitted.Tag = IsolationLevel.ReadCommitted
    RbtReadUncommitted.Tag = IsolationLevel.ReadUncommitted
    RbtRepeatableRead.Tag = IsolationLevel.RepeatableRead
    RbtSerializable.Tag = IsolationLevel.Serializable
    RbtSnapshot.Tag = IsolationLevel.Snapshot
    RbtUnspecified.Tag = IsolationLevel.Unspecified

    RbtUnspecified.Checked = True
End Sub
```

Pour utiliser ultérieurement le niveau de visibilité choisi, on déclare dans le formulaire un niveau d'isolation. Puis nous le valorisons dans le gestionnaire d'évènement présenté ci-dessous :

```
// C#  
  
IsolationLevel oNiveauIsolationDonnees;  
  
private void RbtIsolationDonnees_CheckedChanged(object sender, EventArgs e)  
{  
    oNiveauIsolationDonnees = (IsolationLevel)((RadioButton)sender).Tag;  
}
```

```
// VB .NET  
  
Private Sub RbtIsolationDonnees_CheckedChanged (ByVal sender As  
System.Object, ByVal e As System.EventArgs) Handles RbtChaos.CheckedChanged  
    oNiveauIsolationDonnees = CType(CType(sender, RadioButton).Tag,  
IsolationLevel)  
End Sub
```

Ce gestionnaire d'évènement doit être abonné à l'évènement *CheckedChanged* de tous les contrôles radio boutons désignant un niveau d'isolation de données.

### 2.3.3 Gestion des données

Dans les différentes méthodes du formulaire, nous allons manipuler un *DataAdapter* et une table de données. Nous déclarons alors les deux attributs suivants :

```
// C#  
  
SqlDataAdapter oDataAdapter;  
DataTable oTable;
```

```
// VB .NET  
  
Dim oDataAdapter As SqlDataAdapter  
Dim oTable As DataTable
```

Pour afficher les informations concernant les stagiaires, nous implémentons l'évènement *Click* sur le bouton *Rafraîchir* :



```
// C#

private void CmdRafrachir_Click(object sender, EventArgs e)
{
    // Variables locales.
    SqlConnection oConnexion;
    SqlTransaction oTransaction = null;
    SqlCommand oCommande;

    try
    {
        // Création de la connexion.
        oConnexion = new SqlConnection(@"Data Source=localhost\SQL2008;
Initial Catalog=DotnetFranceA; integrated security=true;");

        // Ouverture de la connexion.
        oConnexion.Open();

        if (ChkUtiliserTransaction.Checked)
        {
            // Création de la transaction.
            oTransaction =
oConnexion.BeginTransaction(oNiveauIsolationDonnees);
        }

        // Création de la commande.
        oCommande = new SqlCommand("SELECT * FROM Stagiaire", oConnexion,
oTransaction);

        // Création et paramétrage du DataAdapter.
        oDataAdapter = new SqlDataAdapter(oCommande);
        SqlCommandBuilder oCommandBuilder = new
SqlCommandBuilder(oDataAdapter);

        // Création de la table de données.
        oTable = new DataTable("Stagiaire");

        // Exécution de la requête et remplissage de la table de données.
        oDataAdapter.Fill(oTable);

        // Affichage des données.
        LstStagiaires.DataSource = oTable;
    }
    catch (Exception aEx)
    {
        MessageBox.Show(aEx.Message);
    }
}
```

```
// VB .NET

Private Sub CmdRafraichir_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles CmdRafraichir.Click
    ' Variables locales.
    Dim oConnexion As SqlConnection
    Dim oTransaction As SqlTransaction = Nothing
    Dim oCommande As SqlCommand

    Try
        ' Création de la connexion.
        oConnexion = New SqlConnection("Data Source=localhost\SQL2008;
Initial Catalog=DotnetFranceA; integrated security=true;")

        ' Ouverture de la connexion.
        oConnexion.Open()

        If (ChkUtiliserTransaction.Checked) Then
            ' Création de la transaction.
            oTransaction =
oConnexion.BeginTransaction(oNiveauIsolationDonnees)
        End If

        ' Création de la commande.
        oCommande = New SqlCommand("SELECT * FROM Stagiaire", oConnexion,
oTransaction)

        ' Création et paramétrage du DataAdapter.
        oDataAdapter = New SqlDataAdapter(oCommande)
        Dim oCommandBuilder As New SqlCommandBuilder(oDataAdapter)

        ' Création de la table de données.
        oTable = New DataTable("Stagiaire")

        ' Exécution de la requête et remplissage de la table de données.
        oDataAdapter.Fill(oTable)

        ' Affichage des données.
        LstStagiaires.DataSource = oTable
    Catch aEx As Exception
        MessageBox.Show(aEx.Message)
    End Try
End Sub
```

Pour enregistrer les modifications dans la base de données :

```
// C#  
  
private void CmdEnregistrer_Click(object sender, EventArgs e)  
{  
    try  
    {  
        if (oDataAdapter != null)  
        {  
            oDataAdapter.Update(oTable);  
        }  
    }  
    catch (Exception aEx)  
    {  
        MessageBox.Show(aEx.Message);  
    }  
}
```

```
// VB .NET  
  
Private Sub CmdEnregistrer_Click(ByVal sender As System.Object, ByVal e As  
System.EventArgs) Handles CmdEnregistrer.Click  
    Try  
        If oDataAdapter Is Nothing Then  
            oDataAdapter.Fill(oTable)  
        End If  
        Catch aEx As Exception  
            MessageBox.Show(aEx.Message)  
        End Try  
End Sub
```

Pour annuler les modifications effectuées depuis le chargement des données ou le dernier enregistrement :

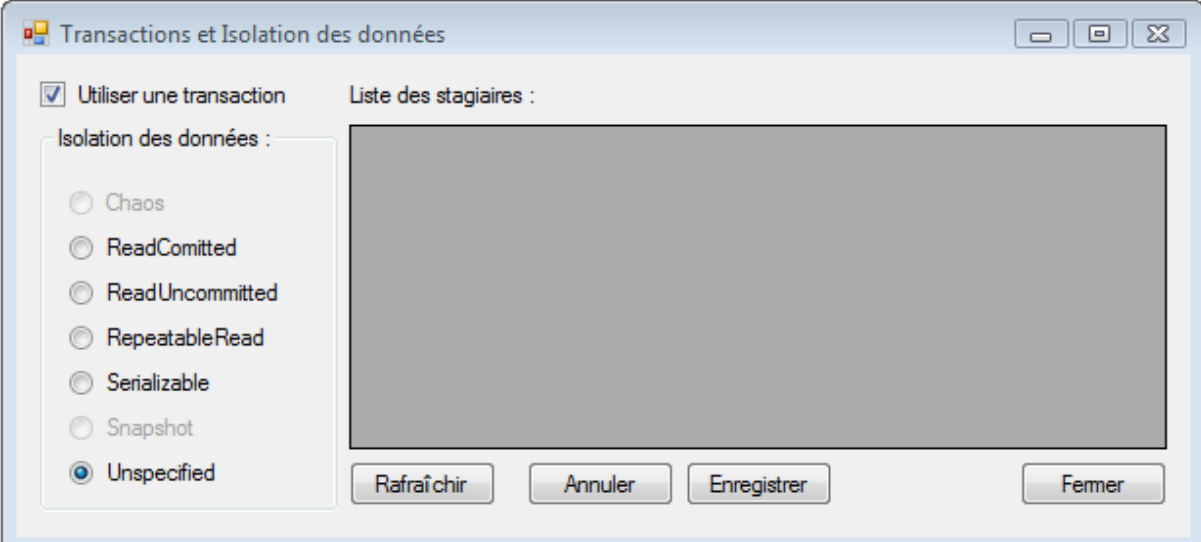
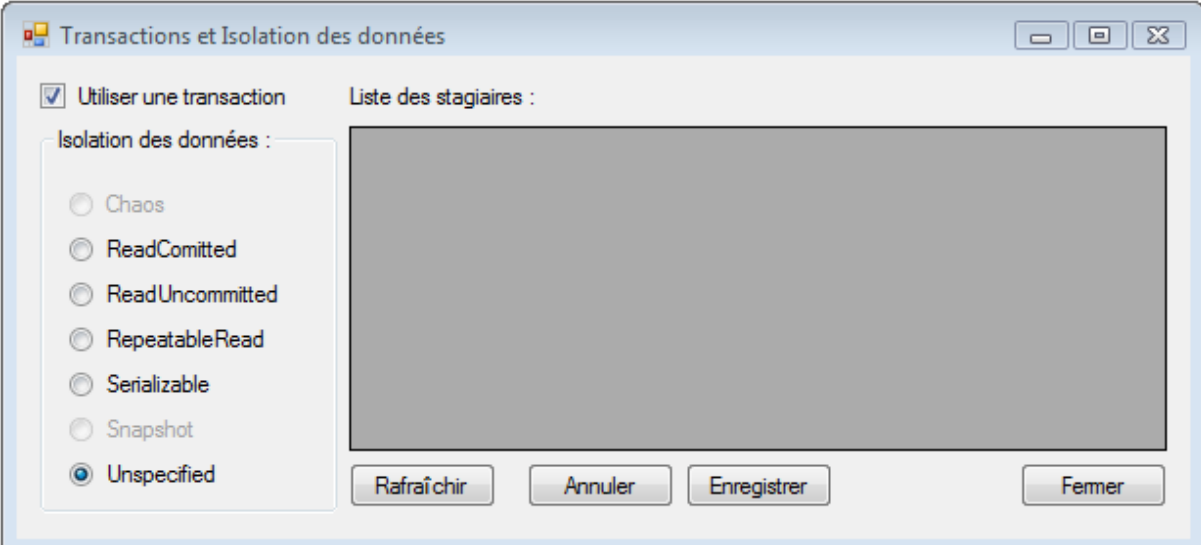
```
// C#  
  
private void CmdAnnuler_Click(object sender, EventArgs e)  
{  
    try  
    {  
        if (oTable != null)  
        {  
            oTable.RejectChanges();  
        }  
    }  
    catch (Exception aEx)  
    {  
        MessageBox.Show(aEx.Message);  
    }  
}
```

```
// VB .NET

Private Sub CmdAnnuler_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles CmdAnnuler.Click
    Try
        If oTable Is Nothing Then
            oTable.RejectChanges()
        End If
    Catch aEx As Exception
        MessageBox.Show(aEx.Message)
    End Try
End Sub
```

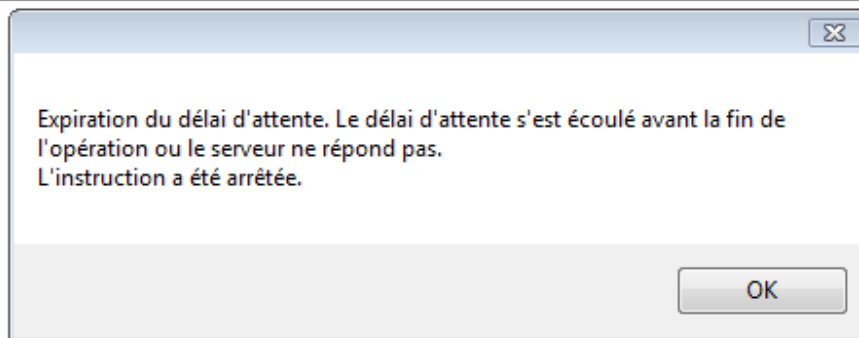
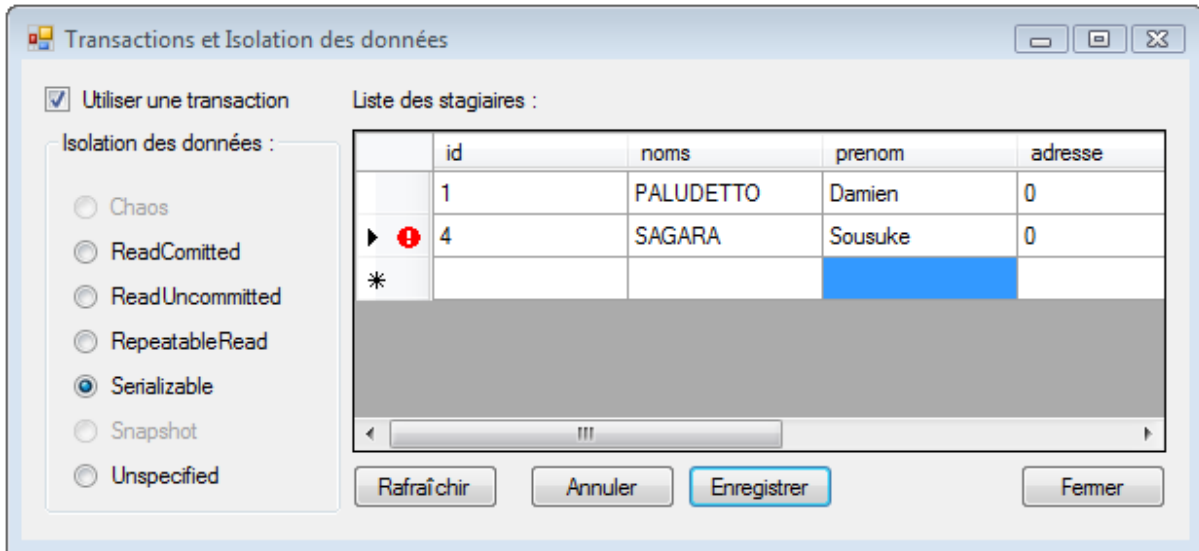
### 2.3.4 Exécution de l'application

Pour mettre en évidence l'utilisation de notre transaction, il faut lancer deux instances de l'application. Pour ce faire, nous allons nous positionner dans le répertoire bin\debug de notre projet, et exécuter deux fois le fichier *ADO\_Transactions\_CS.exe*. Deux formulaires apparaissent alors :



Dans la première instance (celle du haut), on sélectionne le niveau d'isolation de données « Serializable ». Pour rappel, les données peuvent être lues, mais ne peuvent être modifiées. Puis on clique sur le bouton *Rafraîchir*. Les données apparaissent.

Dans la seconde instance (celle du bas), on clique sur le bouton *Rafraîchir*. Puis on modifie le prénom d'un stagiaire, et on clique sur le bouton *Enregistrer* :

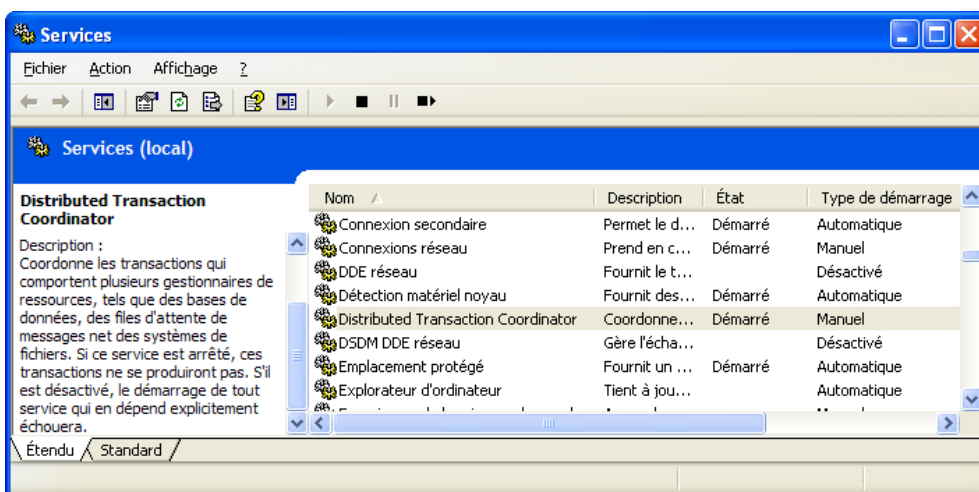


On remarque qu'une exception est levée. Le message d'erreur affiché montre qu'un *Timeout* est survenu. L'enregistrement des données n'a pu être effectué. Nous pouvons donc constater que les données ont bien été verrouillées en écriture par la transaction de la première instance de l'application.

## 3 Les transactions distribuées

### 3.1 Présentation

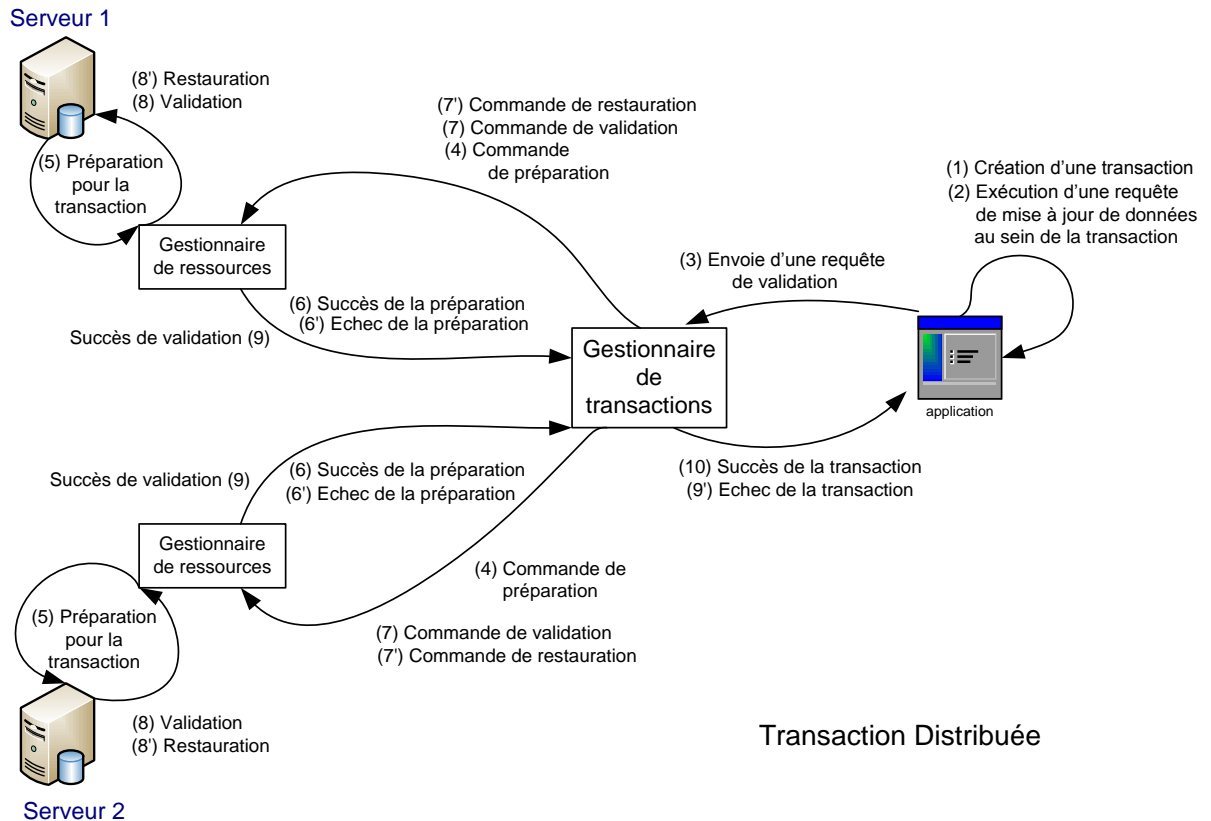
Une transaction distribuée, est une transaction qui fait appelle à plusieurs ressources. Pour cela la transaction utilise des gestionnaires de ressources qui sont eux même gérés par un gestionnaire de transaction. Comme gestionnaire de transaction on utilisera le DTC (Distributed Transaction Coordinator). Ce dernier peut nécessiter d'être démarré manuellement, sans quoi une erreur pourrait se produire lors de la compilation de l'exemple de transaction distribuée que l'on verra dans la partie 3.2. Pour cela allez dans les Services (tapez Services.msc dans cmd), faites un clique droit sur le service en question et cliquez sur démarrer.



Lorsqu'une transaction est amorcée dans une application, une requête de validation est envoyée au gestionnaire de transaction. Ce dernier enverra une commande de préparation à tous les gestionnaires de ressources de la transaction. Après traitement de la commande les gestionnaires de ressources enverront à leur tour un message d'échec ou de succès de préparation au gestionnaire de transaction.

- Si un message d'échec a été envoyé par *au moins un* gestionnaire de ressources, le gestionnaire de transaction envoie alors une commande de restauration à tous les gestionnaires de ressources, puis envoie un message d'échec de la transaction à l'application.
- Si seulement des messages de succès ont été envoyés, le gestionnaire de transaction envoie alors une commande de validation à tous les gestionnaires de ressources. Après validation le gestionnaire de ressource confirme la validation au gestionnaire de ressource, qui à son tour confirme le succès de la transaction à l'application.

Comme le montre le schéma, ce système de validation à deux phases permet de s'assurer que la transaction respecte les propriétés ACID, avec aucune différence de traitement des commandes par un gestionnaire de ressources comparé à un autre.



### 3.2 Mise en œuvre

Pour créer une transaction distribuée on utilise l'espace de nom *System.Transactions*. Il ne faut pas oublier d'ajouter la référence correspondante (*System.Transactions.dll*).

L'espace de nom *System.Transactions* permet d'utiliser l'objet *TransactionScope*. Par défaut *TransactionScope* crée une transaction simple, cependant cette dernière pourra évoluer vers une transaction distributive, c'est le cas lorsque la transaction fera appel à plus d'une base de données (voir l'exemple).

On utilise le mot clé *Using* pour créer un objet de type *TransactionScope*, la méthode *Dispose* sera alors générée automatiquement à la fin de celui-ci. La méthode *Complete* permet de valider la transaction si les commandes ont bien été exécutées. Dans le cas contraire, la transaction est annulée et la base de données retrouve son état initial.



```
//C#
string connectionString1 = "Data Source=NORBERT\\SQLEXPRESS;Initial
Catalog=DotnetFrance;Integrated Security=true";
string connectionString2 = "Data Source=NORBERT\\SQLEXPRESS;Initial
Catalog=dnFrance;Integrated Security=true";

using (TransactionScope Transaction = new TransactionScope())
{
    using (SqlConnection connection1 = new SqlConnection(connectionString1))
    {
        try
        {
            SqlCommand commande = connection1.CreateCommand();
            commande.CommandText = "DELETE FROM Stagiaire WHERE nom like
'PIERRE'";
            connection1.Open();
            commande.ExecuteNonQuery();
            connection1.Close();

            using (SqlConnection connection2 = new
SqlConnection(connectionString2))
            try
            {
                SqlCommand commande2 = connection2.CreateCommand();
                commande2.CommandText = "DELETE FROM Manager WHERE nom like
'Tartonpion'";
                connection1.Open();
                commande.ExecuteNonQuery();
                connection2.Close();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }

    Transaction.Complete();
}
```



```
// VB.net
Dim connectionString1 As String = "Data Source=NORBERT\SQLEXPRESS;Initial
Catalog=DotnetFrance;Integrated Security=true"
Dim connectionString2 As String = "Data Source=NORBERT\SQLEXPRESS;Initial
Catalog=dnFrance;Integrated Security=true"

Using Transaction As New TransactionScope()

    Using connection1 As New SqlConnection(connectionString1)
        Try
            Dim commande As SqlCommand = connection1.CreateCommand()
            commande.CommandText = "DELETE FROM Stagiaire WHERE nom
like 'PIERRE';"
            connection1.Open()
            commande.ExecuteNonQuery()
            connection1.Close()

            Using connection2 As New
SqlConnection(connectionString2)
                Try
                    Dim commande2 As SqlCommand =
connection2.CreateCommand()
                    commande2.CommandText = "DELETE FROM Manager
WHERE nom like 'Tartonpion';"
                    connection1.Open()
                    commande.ExecuteNonQuery()
                    connection2.Close()
                Catch ex As Exception
                    MessageBox.Show(ex.Message)
                End Try
            End Using
        Catch ex As Exception
            MessageBox.Show(ex.Message)
        End Try
    End Using
    Transaction.Complete()
End Using
```

## 4 Conclusion

Ce cours vous a permis d'utiliser les transactions proposées par le Framework .NET, dans l'accès aux données contenues dans une base de données. Trois points essentiels sont à retenir :

- L'utilisation des transactions permet d'exécuter un lot de requêtes en respectant les propriétés ACID.
- L'utilisation des transactions pour verrouiller des données.
- L'utilisation des transactions distribuées.